**David Rosales UTEID: dar542**

**Amy Reed UTEID: alr2434**

**June 17, 2015**

**HW1: Q3-4**

**Question 3a:**

**Peterson’s Psuedocode**

Int turn = 0;

Boolean wantCS[] = {false,false}

requestCS()

wantCS[i] = true;

turn = 1-i

while(wantCS[1-i] && turn == 1-i)

skip(); //while it’s the other guys turn and he wants it , wait

releaseCS()

wantsCS[i] = false

wantCS[1] wantCS[0] turn

**Solution1:**

Ex#1: true false 1 => Set turn = 1 and wantCS[1] to true so I want it and its

my turn but I am still waiting for the other guy to want

it. He doesn’t, so wait.

true true 0 => Other guy wants it, so he sets wantsCS[0] to true and

I go because now turn ==1 and wantCS[0] is true, i.e. the other guys wants it and I was waiting for that. Now he sets turn to himself, so for him both wantCS[1] and turn == 1, so he also goes and we both end up in the CS.

**Solution 2:**

Assume p0 is in the CS and p1 requests to access the CS  
  
    p1 sets  
    j = 0  
    wantCS = {true, true}  
    turn = 1  
  
    p1 never waits for p0 to exit the CS since the condition for the while loop is never met since turn != j.

**Question 3b:**

**Solution 1:**

Assume process 1 requests the CS shortly after process 0 requests the CS  
  
    p0 sets j = 1  
    p0 sets turn = 1  
    p0 sets wantCS = {true, false}  
    before p0 checks the condition on the while loop  
        p1 sets j = 0  
        p1 sets turn = 0  
    p0 checks while loop condition wantCS[0] = true and turn = j so p0 waits  
    p1 sets wantCS = {true, true}  
    p1 checks while loop condition wantCS[0] = true and turn = j so p1 waits  
    a deadlock has been created and both processes continue to wait indefinitely

**Question 4:**

**Bakery Algorithm**

choosing[myId] = true;

for(int j=0; j<processes; j++) {

if(number.get(j) > number.get(myId)) {

number.set(myId,number.get(j));

}

}

number.set(myId, (number.get(myId)+1));

choosing[myId] = false;

for(int j=0; j<processes; j++) {

while(choosing[j]) {

sleep();

}

while((number.get(j) != 0) &&

((number.get(j) < number.get(myId)) ||

((number.get(j) == number.get(myId)) && j < myId))) {

Sleep()

}

}

**Solution:** Execution without choosing[]

Process 1 and 2 start choosing at the same time.

Process 2 picks number 1 and starts checking processes numbers.

Process 2 checks process 1 number and sees that it is == 0 and enters CS because there is no choosing

variable to let process 2 know that number is in flux.

Process 1 picks number 1 and starts checking processes numbers.

Process 1 sees process 2 number == 1, but process 2 id is not lower than process 1 is so process 1 also enters the CS.

Two processes are now in critical section, and mutex doesn’t work.

Explanation for the above scenario is that the 2nd process enters the CS before process 1. The bakery algorithm lets processes enter the CS in a specific order, and in the event of a tie uses the lowest process ID to break the tie. The choose variable is meant to hold a higher number process off from proceeding until the lower number processes have for certain chosen their number avoiding the scenario where the higher number process would miss detecting the tie.  
  
To restate the above, pr 1 and 2 are in line and request entry to the bakery at the same time. Process 1 should be served before process 2 in this case. However, if there is no choosing variable, a possible could be that process 1 and 2 both get the same number and process 2 chooses the number before process 1 and is allowed to enter the CS before process 1, and then process 1 also enters the CS.  
  
With the choosing variable in place, even if process 2 finishes choosing before process 1 in the scenario above, he must wait for process 1 to finish choosing still before entering the CS.